Information Retrieval

H. KOLLER, Editor

An Online Editor

L. Peter Deutsch and Butler W. Lampson
University of California,* Berkeley, California

An online, interactive system for text editing is described in detail, with remarks on the theoretical and experimental justification for its form. Emphasis throughout the system is on providing maximum convenience and power for the user. Notable features are its ability to handle any piece of text, the content-searching facility, and the character-by-character editing operations. The editor can be programmed to a limited extent.

Introduction

One of the fundamental requirements for many computer users is some means of creating and altering *text*, i.e., strings of characters. Such texts are usually programs written in some symbolic language, but they may be reports, messages, or visual representations. The most common means of text handling in the last few years has been the punched card deck. Paper tape and magnetic tape have also been used as input, but the fact that individual cards can be inserted and deleted manually in the middle of a deck has made the keypunch the most convenient and popular device for text editing.

With the appearance of online systems, however, in which text is stored permanently within the system and never appears on external storage media, serious competition to the punched card has arisen. Since the online user has no access to his text except through his console and the computer, a program is needed, however minimal, to allow him to create and modify the text. Such a program is called an *editor* and presupposes a reasonably large and powerful machine equipped with disk or drum storage. The user therefore gains more convenience than even the most elaborate keypunch can provide. The characteristics of a good online editor and some of the techniques which can be used in implementing it are the subjects of this paper.

The number of such editors known to the author is not large. Time-sharing systems on the 7094 at Project MAC and the AN/FSQ32 at the System Development Corporation [2] have them, and at least two have been written for the PDP-1 [6]. The present paper is built around a description of the editor in the Berkeley time sharing system for the SDS-930 [3, 4], which is called QED. An attempt is made to discuss all the valuable features which have been built into editors for teletypes or typewriters, with the exception of the "runoff" [1] facility for producing properly formatted final documents. Systems for CRT displays are not considered, since many of their design considerations are quite different.

The most important characteristic of an editor is its convenience for the user. Such convenience requires a simple and mnemonic command language, and a method of text organization which allows the user to think in terms of the structure of his text rather than in some framework fixed by the system. In view of the speed and characteristics of a teletype, there are substantial advantages to a line-oriented system. However, the physical mechanism of the teletype makes it difficult to deal with individual characters, and the speed makes a larger unit somewhat inconvenient.

Fortunately, line orientation does not impose any restrictions on the text which can be handled by the editor, since all forms of text fall of necessity into lines. Preservation of this generality is one of the important design criteria, and it will be seen from the description below that very little has to be sacrificed to this end.

In order to allow the user to orient himself in the text, the concept of *searches* or *content addressing* has been introduced. In its simplest form, a content address allows the user to reference the lines of his text by the labels which he has naturally attached to them in the course of the construction. More general searches allow him to find occurrences of specified strings of characters. This device allows the user a maximum amount of freedom to arrange his text as he sees fit without compromising his ability to address it. Many particular conventions can be accommodated within the general framework.

The basic characteristics of QED are line organization and content addressing. For the casual user of the system, four or five commands and understanding of the addressing scheme will provide ample power. A more frequent user will, however, be able to make good use of additional features, which include: (1) a *line editing* mode which permits character-by-character editing of a line; (2) buffers for storing frequently used text or command sequences; (3) a substitute command; (4) the ability to save a set of editing commands and later repeat the edit automatically; (5) automatic adjustable tab stops.

Another important consideration in the design of QED

* Project Genie. The work described in this paper was supported by the Advanced Research Projects Agency of the Department of Defense under Contract SD-185.

has been simplicity of implementation. The original version of the system, admittedly without many of the elaborate features, was designed, written, and debugged by one man in less than a week, and the entire program now occupies less than 4000 words of reentrant code.

Basic Editing Operations

QED regards the text on which it is operating as a single long string of characters called the *main text buffer*. Structure is imposed on this string by the interpretation of carriage returns as line delimiters. Lines can be addressed by absolute line number, and the characters on line n are those between the $(n-1)$st and the nth carriage returns, including the latter but excluding the former. The line number of a particular line may, of course, change if carriage returns are added or deleted earlier in the buffer. These absolute line numbers are in principle sufficient, together with three simple commands, for any editing operation. All the other devices for addressing text are syntactically equivalent to line numbers; i.e., any address can be replaced by the line number of the line it addresses. It will be convenient in the remainder of this section to take advantage of this fact and defer discussion of other addressing mechanisms until the basic commands have been presented.

The normal state of the editor is its *command* mode. Whenever this mode is entered, it prints a carriage return and a "*" to indicate its readiness for a command. The other modes are *text* mode and *line edit* mode; they will be explained in turn.

All commands take one of the following forms:

⟨command⟩
⟨address⟩⟨command⟩
⟨address⟩,⟨address⟩⟨command⟩

Some commands also take additional arguments. The command itself is in most cases specified by a single letter. The time sharing system in which the editor runs allows programs to interact with the teletype on a character-by-character basis, and the QED command recognizer makes use of this capability to supply the remaining letters of the command. This has proved to be a valuable aid to the beginning and to the occasional user of the editor. An expert user can suppress the command completion.

After a command has been given, it must be confirmed by a period. The teletypes used in the system are full duplex, so that the period may be typed in while the command is being completed. It is therefore unnecessary for the user to synchronize his typing with the computer's responses.

The three basic editing commands are INSERT, DELETE, and PRINT. An insert command has the form

*12INSERT.

The computer generates a carriage return and goes into text mode, in which it will accept a string of characters to be inserted in the text immediately preceding line 12. The text is terminated by a teletype *control character*, control **D**, which is generated by holding down the CONTROL shift key and pushing the "D" key. (Control characters appear in boldface type in this paper.)

The existence of control characters, which do not, with a few exceptions, produce any effect on the teletype, makes it possible for the user to give instructions to the editor while he is in text mode without any escape character convention. In addition to **D** for terminating text input, three delete characters are available. **A** deletes the last character typed which has not already been deleted and **Q** the last line. The third delete character, **W**, deletes a word, which is defined as all the immediately preceding blanks and all the characters up to the next preceding blank. These characters permit the immediate correction of minor errors in text input.

Although the delete characters themselves are nonprinting, it is desirable that something should appear on the paper when they are used, since otherwise it becomes very difficult to keep track of the state of the text being entered from the keyboard. It has therefore been arranged that **A** will cause a "↑" to be printed, **Q** a "←", and **W** a "\". This convention has the unfortunate result that text containing these characters becomes confusing when the delete characters are used. No confusion is possible, however, when the text is typed out by the editor.

Another important feature of QED's text mode is the tab stop. The user can set tab stops to any positions he desires, using the TABS command. For example:

*TABS.
5, 10, 15, 20.

After the command has been given, there are tab stops at positions 5, 10, 15, and 20 on the line. Thereafter the character **I** (labeled TAB on the keyboard) will generate enough blanks to bring the printing element of the teletype to the next tab stop.

A command complementary to INSERT is DELETE, which takes the form

*12DELETE.

and causes line 12 to be deleted from the text. Another form is

*12,14DELETE.

which causes lines 12, 13, and 14 to be deleted.

These two commands are sufficient for any editing operation. The third basic QED command is PRINT, which may also be given with one or two addresses.

*12PRINT.

prints line 12.

*12,14PRINT.

prints lines 12, 13, and 14. When a line is prin[illegible] ing characters (those in the teletype type box) [illegible] in their proper sequence. Nonprinting char[illegible] control characters, are printed as

&(letter corresponding to the control cha[illegible]

An editor must also be able to read in data from some storage medium and write out data for later use. These functions are provided in QED by READ and WRITE commands, which take the form

*READ FROM PROG1.

and

*WRITE ON PROG1.

The READ command appends the contents of the file to the main text buffer. The WRITE command may be preceded by two line numbers, in which case only the specified portion of the text will be written out.

Figure 1 illustrates the creation and correction of a small program using the commands which have just been described. It also makes use of the APPEND command, an INSERT which puts the new text after the line addressed. An APPEND with no argument puts the text at the end of the buffer. Perusal of this example will suggest the utility of a number of additional conveniences in the editor. The simplest of these is the CHANGE command, which combines the functions of INSERT and DELETE. In fact,

*12CHANGE.

is exactly equivalent to

*12DELETE.

*12INSERT.

Like DELETE, CHANGE can be used with two line numbers. The number of lines inserted has no relation to the number of lines deleted.

Two minor extensions of PRINT are single-character commands to print the next line of text (line feed) and to print the preceding line (↑). There is also a command which prints the text in pages 11 inches long; it provides page headings and numbers if requested.

In the original implementation of QED, the main text buffer was stored as a string of consecutive characters in memory. This simple storage allocation scheme makes it easy to implement the commands so far discussed. A deletion, for example, is accomplished by moving the character following the deleted section towards the beginning of the buffer to cover the ones being deleted. See Figure 2.

Insertion is slightly more complex. The text to be inserted is collected in a special storage area. When it has been completely typed in, the characters after the point at which the insertion is to be made are moved far enough toward the end of the buffer to make room for the new text, which is copied into the space created for it.

Only three pointers to the text are maintained by the system: one to the beginning of the buffer, one to the end, and one to the current line. This means that no readjustment of pointers is required by the displacements described above.

When the amount of text being edited becomes large, the simple algorithms begin to become unattractive in terms of efficiency. This problem can be alleviated, however, by dividing the text into artificial pages and leaving

```
*APPEND.
10  REARA↑D, 100, N
     SUM = 0
     DO 20 I = 1, 1, N
D
*1DELETE.
*1INSERT.
10 READ 100, N
D
*APPEND.
    -READ 101, X
20  SUM - SUM Q←
20  SUM = SUM + X
    .WRITE 201, S, W\101, SUM
100 FORMAT (6I)
101 FORMAT (F10.5)
    END
D
*7DELETE.
*7INSERT.
100  FORMAT (I6)
D
*3DELETE.
*3INSERT.
DO 20 I = 1, N, 1
D
*1, 9PRINT.
10  READ 100, N
    SUM = 0
    DO 20 I = 1, N
    READ 101, X
20  SUM = SUM + X
    WRITE 101, SUM
100 FORMAT (I6)
101 FORMAT (F10.5)
    END
```

FIG. 1. Example of basic QED commands. Note that control characters (in boldface here) do not print anything

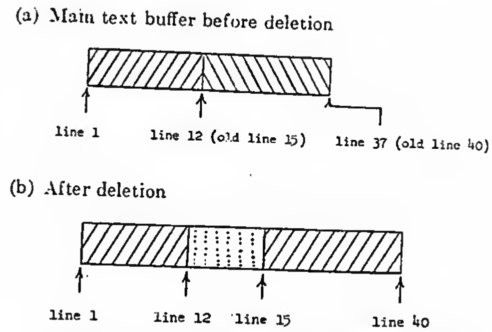

FIG. 2. Action of the command *12,14DELETE

a reasonable amount of free space at the end of every page. The effect of nearly all the displacements discussed above can then be confined to a single page. When large insertions or deletions are made it may be necessary to redo the paging completely, but this is an infrequent occurrence. Such a paging scheme is further recommended by the fact that it permits most of the text to be kept out of main memory most of the time. Only one or two pages need to be available for any single editing operation.

Efficiency can be further increased, in a machine which is basically word-oriented, by storing each line in an integral number of words. Since the line always ends with exactly one carriage return, the last word can be filled out if necessary with additional carriage returns without any possibility of confusion being introduced. This arrangement greatly speeds up most searches and all insertions or deletions, since the text can now be handled a word at a time. It may also be convenient to keep the number of words in each line at the beginning of the line.

All these improvements have been incorporated in the latest implementation of QED. The result has been that most editing operations, even on files of 50 or 100 thousand characters, can be done with less than a tenth of a second of computation.

Addressing

As we have already noted, and as even the trivial example in Figure 1 suggests, absolute line numbers are not a sufficiently powerful addressing mechanism. An attempt to edit a 1000-line program would illustrate this point even more forcibly. It is necessary to be able to address a line by its contents as well as by its location. The simplest way to arrange this is to provide each line with a *sequence* number, generated either automatically by the editor or manually by the user. The lines are kept ordered by sequence number and can be addressed directly. There are two objections to this scheme.

(1) It requires the user to concern himself with an artificial device which has no relevance to his text but nonetheless intrudes on it, wasting space and time on output and reducing its usefulness as a document.

(2) Insertions and deletions will eventually force renumbering of the lines. When this happens, a complete new listing must be generated if the sequence numbers are to be of any use. Furthermore, as a result of this process numbers do not stay attached to lines.

A more satisfactory scheme is a more general kind of content addressing. In its simplest form this allows the user to refer to the line

```
XYZ  ADD  =14
```

with the address :XYZ:. The meaning of this construction is that the text is to be searched for a line beginning with the characters inside the colons, with the requirement that they be followed by a character which is not a letter or digit. A line such as

```
XYZA  SUB  =24
```

will therefore not be found. The search begins with the line after the one last accessed and continues, cycling to the beginning of the buffer if it runs off the end, until a line beginning with the specified string is found, or until the entire buffer has been scanned. In the latter case, QED prints "?" and awaits a new command.

This kind of content addressing, called *label* addressing, is convenient for many kinds of text, including most programs. It is also possible, however, to search for a line containing any string of characters in any position by using the construct [⟨string⟩], where ⟨string⟩ refers to any string of characters not containing "]".

The usefulness of content addresses is enhanced by the fact that they may be followed by integer displacements, positive or negative. Thus in Figure 1, the third line could be addressed in any of the following ways:

3	
6−3	
10−9+2	
:10:+2	since :10: refers to line 1
:20:−2	since :20: refers to line 5
[I=1]	since only line 3 contains the string "I=1"
[101, SUM]−3	since only line 6 contains the string "101, SUM"

The search can be started at any line, rather than at the current one, by putting the starting line immediately before the search construct. Thus in Figure 1, 4[I] would find line 6, as would :20: [101].

Two minor devices offer additional convenience. The character "." refers to the current line and the character "$" to the last line in the buffer. The "current line" is defined according to rigid rules which are set forth in the listing of Table I. The reason for this careful specification is that an experienced user of the editor makes frequent use of "." in performing insert and delete operations. If he cannot be perfectly sure of its value, he is forced to print the lines he intends to work on before doing the edits, which is very time-consuming.

TABLE I. Rules for Determining the Value of "."

Last operation performed	Value of "."
Successful search	Line found
Unsuccessful search	Unchanged
Any insertion	Last line inserted
Any deletion	Line preceding first deleted line
Print or write	Last line printed or written

Another very useful convention is that "." is assumed as the argument of a command which is given without one. Thus PRINT. will print the current line. Exceptions to this rule are READ and APPEND, which assume "$" unless told otherwise, and WRITE, which assumes "1, $".

Two minor commands permit an address to be displayed either as an absolute line number (=command) or in symbolic form, as the label of the nearest preceding line which does not begin with a blank or asterisk, followed by an integer displacement (←command). Thus with the [illegible]

```
old line                    BUSIE OLD FOOLE, UNRULY SUNNE
control characters input    CCC SCCCCCCCCCSCCCCZ        *
ordinary characters input      Y                N
output during edit          BUSY% OLD FOOL,% UNRULY SUN
new line                    BUSY OLD FOOL, UNRULY SUN

old line                    WHY DO  YOU  THUS,
control characters input    Z     X    E         D
ordinary characters input   O     U      ,  ST THOU
output during edit          WHY DO%%%%<ST THOU THUS,
new line                    WHY DOST THOU   THUS,

old line                    THRU   WINDOWS AND THRU CURTAINS CALL ON US?
control characters input    CCCSE        Z      . C C  T
ordinary characters input          OUGHO          E ,
output during edit          THR%<OUGH WINDOWES,
remainder of old line                         AND THRU  CURTAINS CALL ON US?
new line so far             THROUGH WINDOWES,
control characters input                      Z         C   Z          D
ordinary characters input                     R        O  GHN         E
output during edit          THROUGH WINDOWES, AND THROUGH CURTAINES CALL ON US?
new line                    THROUGH WINDOWES, AND THROUGH CURTAINES CALL ON US?
```

Fig. 3. Examples of line edits. A * in the control character lines is used to indicate carriage return

Figure 1 in the buffer, QED would respond to :10:= with 1, to :100:= with 7, to 6← with :20:+1, to [SUM+X]−1← with :10:+3.

Line Editing

Circumstances frequently arise in which it is necessary to make small changes to a line already in the text: two or three characters may need to be inserted or deleted. This is an area in which the weaknesses of the teletype make themselves felt, and a truly satisfactory solution can only be had with a display device on which the user can point to the characters he wishes to change. There are, unfortunately, no mechanisms for addressing characters within a line on a teletype which are not more trouble than they are worth.

QED does, however, contain a character editing mechanism which provides many of the features a user might want. This power has been purchased at the cost of considerable complexity; although the basic idea of the line edit is simple, there is a profusion of commands to speed up the handling of special cases which is somewhat bewildering to the new user.

The command *12EDIT. will cause line 12 to be typed out, followed by a carriage return. The editor is now in its line edit mode, in which it will recognize a number of teletype control characters in addition to the **A**, **Q**, **W**, and **D** which are normally recognized when text is being typed in. The new characters are interpreted as instructions for the construction of a new line from the old one which was typed out. These instructions cause characters to be *copied* from the old line into the new one, *skipped* over without being copied, *replaced* or *inserted*. When the new line is complete, it will replace the old one, and QED will return to command mode. The simple examples in Figure 3 will clarify the process.

The first example shows the use of **C** to copy characters and **S** to skip over them. Note that when a character is skipped, a "%" is printed so as to keep the new line aligned with the old one. When an ordinary character is typed in, it replaces the corresponding character in the old line. To save repetition of **C**, a **Z** causes the old line to be copied up to and including the next occurrence of the following character. Note that the latter is not printed when it is typed in, but when it is reached in the line. This is accomplished by suppressing the echo for the character after the **Z**, another application of the full-duplex capabilities of the teletype. The result is that the edited line continues to be properly aligned with the old one.

The second example illustrates the use of **X** to skip to the next occurrence of a character; this instruction is exactly analogous to **Z**. Also shown is the insertion of characters: an **E** causes ordinary characters to be inserted rather than replace characters already existing. The **E** causes a "<" to be printed. A second **E** will switch back to replacing and print a ">". Insertion of course spoils the alignment. It can be restored with a TYPE instruction (**T**), which types the remainder of the old line and the portion of the new line so far constructed, and aligns the ends properly. The third example illustrates this process.

A line edit is usually terminated by a carriage return, which suppresses the remainder of the old line, or by a **D**, which copies the remainder of the old line into the new line. Both are illustrated, in the first and second examples, respectively.

Figure 4 is a list of the control characters recognized in a line edit. The ones dealing with tabs are useful for editing one field of a fixed-field line. Illegal instructions, such as **Z** followed by a character not in the old line, cause the teletype bell to ring and are otherwise ignored. Charac-

ters typed after the whole of the old line has been copied are appended to the new line, regardless of whether the mode is replace or insert. Note that the escape character V allows any character to be added to the text regardless of its normal interpretation as a command. It works throughout the system, not just in the line edit mode.

Figure 5 is a repetition of the edit performed in Figure 1.

Character	*Function*
A	Delete preceding character in new line
X	Delete preceding character in new line. Backspace pointer to old line if deleted character was copied from old line.
W	Delete preceding word
Q	Restore old line
C	Copy character from old to new line
S	Skip character in old line, print %
Z	Copy up to and including the next occurrence of following character
O	Copy up to but not including the next occurrence of the following character
X	Skip up to and including next occurrence of following character, print %
P	Skip up to but not including the next occurrence of the following character, print %
T	Type old and new lines
Y	Copy remainder of old line to new one, without printing, and start edit over on new line
H	Copy to end of old line
I	Tab, i.e., replace old line with blanks up to next tab
U	Copy old line up to next tab
c.r.,	Skip rest of old line and terminate edit
D	Copy and print rest of old line and terminate edit
F	Copy rest of old line without printing and terminate edit

Fig. 4. Line edit instructions

```
•APPEND.
10  BEARA ' D, 100, N
    SUM = 0
    DO 20 I = 1, 1, N
D
•:10:MODIFY.
{ZD        SD
{10 BEAD% 100, N
•APPEND.
    READ 101, X
20  SUM - SUM Q—
20  SUM = SUM + X
    WRITE 201, S, W-101, SUM
100  FORMAT (6I)
101  FORMAT (F10.5)
    END
D
•:100:MODIFY.
{Z(                116D
{100  FORMAT (I6)
•:10: + 2MODIFY.
Z,               N, 1D
DO 20 I = I, N, 1
```

Fig. 5. The example of Figure 1 redone. The lines paired with braces are the editing characters in a line edit (above) and the characters typed out during the edit (below).

The same errors are made, but many of the features described above are used to speed the process. It illustrates the MODIFY command, which is identical to EDIT except that it suppresses the initial printing of the old line.

Substitution

An alternative method of altering a few characters in the middle of a line is the SUBSTITUTE command, which is modeled after the CHANGE command in the Project MAC editor [1]. Its simplest form is illustrated by the following example. Suppose the current line is

NOW IS THE TIME FOR ALL GOOD MEN . . .

Then the user's command

SUBSTITUTE *DAY* ' FOR *TIME*/

would result in

NOW IS THE DAY FOR ALL GOOD MEN . . .

The italicized characters in the command above are the ones supplied by the user. The "/" is the delimiter for this substitution; the delimiter is taken to be the first nonblank character after the initial "S", and it terminates both new and old strings. If command completion had been turned off, the command would have appeared as

S/DAY/TIME.

Figure 6 illustrates the edits which were performed in

```
•PRINT.
BUSIE OLD FOOLE, UNRULY SUNNE
•SUBSTITUTE // FOR /E/
•SUBSTITUTE /Y/ FOR /I/
•SUBSTITUTE /N/ FOR /NN/
•PRINT.
BUSY OLD FOOL, UNRULY SUN
•.+1PRINT.
WHY DO YOU THUS,
•SUBSTITUTE /ST TH/ FOR /Y/
•PRINT.
WHY DOST THOU THUS,
•.+1PRINT.
THRU WINDOWS AND THRU CURTAINS CALL ON US?
•SUBSTITUTE /THROUGH/ FOR /THRU/
•SUBSTITUTE /WES,/ FOR /WS/
•SUBSTITUTE /NES/ FOR /NS/
•PRINT.
THROUGH WINDOWES, AND THROUGH CURTAINES
CALL ON US?
```

Fig. 6. Examples of substitution

Option	*Effect*
:W	Display each substitution bef[illegible] and wait for the user to acc[illegible] it.
:L	Display each substitution afte[illegible]
:⟨decimal number⟩	Terminate the command after [illegible] number of substituti[illegible] made.

Fig. 7. Options for the SUBSTITUTE comm[illegible]

Figure 4 with the line edit as they might be done with SUBSTITUTE.

The substitution may be done for the text in a number of lines in the obvious way. Thus

1,$SUBSTITUTE /+/ FOR /−/

changes every "−" in the main text buffer to a "+". A number of options may also be specified. They take the form

:(option)

and come immediately after the initial "S". Because of this convention, ":" may not be used as a delimiter. The most important options are listed in Figure 7. For example,

1,$SUBSTITUTE :1/ALPHA/ FOR /BETA/

will change the first occurrence of "BETA" in the text buffer to "ALPHA".

String Buffers

Although QED is not a programming language, it does have one feature which makes it possible, among other things, to write simple programs in it, and that is the 36 string buffers, identified by the digits 0 through 9 and the letters A through Z. Each string buffer can be *loaded* with an arbitrary string of characters, either from specified lines of the main text buffer or from teletype input. The buffer can be *called* by the two characters Bn where n is a letter or digit. The editor then behaves *exactly* as though the characters in the buffer were being typed in on the teletype.

This fact has a number of implications. First of all, it permits the user to insert a frequently used phrase in the text by typing just two characters. Secondly, it permits him to move around sections of his text by loading them into string buffers and inserting the buffers at the desired points. To facilitate this operation, a command is available which loads a section of the text buffer into a string buffer and deletes it from the text buffer.

Thirdly, it is possible to put frequently used commands into string buffers. The existence of the escape character, V, which causes the next character to be taken literally no matter what it is, allows control characters such as **A** to appear in buffers. In particular, it allows calls on other buffers to appear in a buffer, and the convention that a call on the buffer itself is taken as a loop command permits simple functions to be performed repeatedly. For example, if the string

.+M.CSSSDBA

is put into buffer A, then a call on this buffer in command mode will cause the second, third, and fourth characters to be removed from every line in the buffer following the current one.

The possibility of extending this elementary programming capability in QED has been seriously considered. The most obvious addition is some kind of conditional facility, and a pattern-matching feature similar to the one in SNOBOL [5] has also received attention. The tentative conclusion has been that such features would be of marginal utility, since small programs can readily be written in SNOBOL or other string-processing languages to accomplish repetitive editing operations.

Several of the numbered buffers are automatically loaded by QED under certain conditions. In particular, the argument of a search is put into buffer 0, any block of text deleted by a DELETE or CHANGE is put into buffer 1 (unless it is too big), as is the text altered by an EDIT or SUBSTITUTE command. This allows an erroneous editing operation to be undone with a small amount of work. Finally, the new and old strings in a SUBSTITUTE command are put into buffers 2 and 3.

Re-editing

It is possible to regard the commands given to QED during an editing session as text which the system can be told to store. At a later time this text can be retrieved and fed back to QED as commands. In this way it is possible to maintain several slightly different versions of a body of text without using up a great deal of storage space, simply by keeping one copy of the text and some small files containing editing commands which create the various versions from the single standard one. In addition to the saving in space, there is the further advantage that changes in the text which are common to all versions need be made only once. Furthermore, since the commands are simply text, they can themselves be modified using QED before they are used to perform the edit. This is sometimes a good way of correcting errors in editing.

Another use for saved commands is in combining several edits into a single one in a normalized form in which the commands are reduced to INSERT, DELETE, and CHANGE and ordered by the absolute line numbers of the lines of text affected. This combination and normalization is not a trivial operation, since it is not in general true that QED commands commute, but it is possible, and a program to accomplish it has been designed. The result is a complete and easily referenced record of all the changes made to a body of text over what may be a long period of time.

Conclusion

We have now considered in some detail all of the significant features which are present in one online editor. QED has been used extensively for about two and a half years, and in its present form reflects the judgment of the group which developed the Berkeley system as to the desirable characteristics of an online teletype text editor. The primary emphasis throughout the design has been on accommodating the system to the needs of the users, both novices and experts. The addressing devices have proved extremely convenient to use on a wide variety of text. These and the line edit are the facilities which attract the most attention, but many other aspects of the editor are of importance to the user, even though he may not

(*Continued on page 803*)

```
      rh[i1] := (rhi1-rhi)/denom;
      i1 := i;  ai1 := ai;  rhi1 := rhi
    end i
  end j;
  comment equate (m+1)th difference to zero to determine h;
  h := -a[mplus1]/rh[mplus1];
  comment with h known, combine the function and deviation
    differences;
  for i := 0 step 1 until mplus1 do
    a[i] := a[i] + rh[i] × h;
  comment compute polynomial coefficients;
  for j := m - 1 step -1 until 0 do
  begin
    xj := rx[j];  i := j;  ai := a[i];
    for i1 := j + 1 step 1 until m do
    begin
      ai1 := a[i1];
      a[i] := ai - xj × ai1;
      ai := ai1;  i := i1
    end i1
  end j;
  comment if the reference deviation is not increasing mono-
    tonically then exit;
  hmax := abs(h);
  if hmax ≤ prevh then
  begin a[mplus1] := -hmax;  go to fit end;
  comment find the index, imax, and value, himax, of the largest
    absolute error for all sample points;
  a[mplus1] := prevh := hmax;  imax := r[0];  himax:= h;
  j := 0;  rj := r[j];
  for i := 1 step 1 until n do
    if i ≠ rj then
  begin
    xi := x[i];  hi := a[m];
    for k := m - 1 step -1 until 0 do
    hi := hi × xi + a[k];
    hi := hi - y[i];  abshi := abs(hi);
    if abshi > hmax then
    begin hmax := abshi;  himax := hi;  imax := i end
  end
  else
  if j < mplus1 then
  begin j := j + 1;  rj := r[j] end;
  comment if the maximum error occurs at a nonreference
    point, exchange this point with the nearest reference point
    having an error of the same sign and repeat;
  if imax ≠ r[0] then
  begin
    for i := 0 step 1 until mplus1 do
    if imax < r[i] then go to swap;
    i := mplus1;
swap: nexthi := if i - i ÷ 2 × 2 = 0 then h else -h;
    if himax × nexthi ≥ 0 then r[i] := imax
    else
    if imax < r[0] then
    begin
      j1 := mplus1;
      for j := m step -1 until 0 do
      begin r[j1] := r[j];  j1 := j end;
      r[0] := imax
    end
    else
    if imax > r[mplus1] then
    begin
      j := 0;
      for j1 := 1 step 1 until mplus1 do
      begin r[j] := r[j1];  j := j1 end;
      r[mplus1] := imax
    end
    else r[i-1] := imax;
    go to start
  end;
fit:
end chebfit
```


CERTIFICATION OF ALGORITHM 91 [E2] CHEBYSHEV CURVE-FIT [Albert Newhouse *Comm. ACM 5* (May 1962), 281; *6* (April 1963), 167; *7* (May 1964), 296]

J. Boothroyd (Recd. 15 May 1967 and 5 Sept. 1967)
University of Tasmania, Hobart, Tasmania, Australia.

In addition to the corrections noted by R. P. Hale [op. cit., April 1963] and P. Naur [op. cit., May 1964], the following changes are necessary:

1. The first statement should be k := *entier*$((m-1)/(n+1))$
2. A semi-colon should precede label $L1$.

With these changes the procedure ran successfully using Elliott 503 Algol.

Although this procedure is an implementation of a finite algorithm, roundoff errors may give rise to cyclic changes of the reference set causing the procedure to fail to terminate.

Algorithm 318 [J. Boothroyd, Chebyshev Curve-Fit (Revised), *Comm. ACM 10* (Dec. 1967), 801] avoids this cycling difficulty, uses less than half the auxiliary array space of Algorithm 91 and, on test, appears to be at least four times as fast.

Deutsch and Lampson—Cont'd from p. 799

be consciously aware of them. Many small details throughout the system have been arranged to permit smooth and rapid operation.

Acknowledgment. The basic framework of QED was designed and the program written by Mr. Deutsch. Many people have contributed suggestions for its improvement; we are especially indebted to M. W. Pirtle and R. Morris in this respect.

Received August, 1966; revised August, 1967

REFERENCES

1. Crisman, P. A. (Ed.). *The Compatible Time-Sharing System: A Programmer's Guide*, 2nd. ed. MIT Press, Cambridge, Mass., 1965, Section AH.9.01.
2. Aranda, S. M. Q-32 Time-sharing system user's guide, executive service: context editing (EDTXT). SDC-TM-2708/204/00, Sys. Devel. Corp., Santa Monica, Calif., Mar. 1966.
3. Lampson, B. W., Lichtenberger, W. W., and Pirtle, M. W. A user machine in a time-sharing system. *Proc. IEEE 54*, 12 (Dec. 1966), 1766–1774.
4. Angluin, D. C., and Deutsch, L. P. Reference manual: QED Time-sharing editor. Project Genie Doc. R-15, U. of California, Berkeley, Calif., Jan. 1967.
5. Farber, D. J., Griswold, R. E., and Polonsky, I. P. The SNOBOL3 programming language. *Bell Sys. Tech. J. 45*, 6 (July 1966), 845–944.
6. Murphy, D. TECO. Memorandum, Bolt, Beranek and Newman, Cambridge, Mass. (Nov. 1966).

:letype braille

: brailler is a simple
·h internal character
orrespond to the left
nd then follows these
cters are then trans-
that the braille col-
vay they will appear
ersion of the braille

Please turn to page 440)

SET WITH BRAILLE
EFT AND RIGHT
S

I	J	K	L	M
CJ	CE	AG	HG	AJ
V	W	X	Y	Z
HN	CH	AA	AH	AL
9	$	(	)	*
NC	NL	HL	LH	JN

Regular Expression Search Algorithm

Ken Thompson
Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

A method for locating specific character strings embedded in character text is described and an implementation of this method in the form of a compiler is discussed. The compiler accepts a regular expression as source language and produces an IBM 7094 program as object language. The object program then accepts the text to be searched as input and produces a signal every time an embedded string in the text matches the given regular expression. Examples, problems, and solutions are also presented.

KEY WORDS AND PHRASES: search, match, regular expression
CR CATEGORIES: 3.74, 4.49, 5.32

The Algorithm

Previous search algorithms involve backtracking when a partially successful search path fails. This necessitates a lot of storage and bookkeeping, and executes slowly. In the regular expression recognition technique described in this paper, each character in the text to be searched is examined in sequence against a list of all possible current characters. During this examination a new list of all possible next characters is built. When the end of the current list is reached, the new list becomes the current list, the next character is obtained, and the process continues. In the terms of Brzozowski [1], this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched. The parallel nature of this algorithm makes it extremely fast.

The Implementation

The specific implementation of this algorithm is a compiler that translates a regular expression into IBM 7094 code. The compiled code, along with certain runtime routines, accepts the text to be searched as input and finds all substrings in the text that match the regular expression. The compiling phase of the implemention does not detract from the overall speed since any search routine must translate the input regular expression into some sort of machine accessible form.

In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. The execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequel characters in the regular expression.

This compile-search algorithm is incorporated as the context search in a time-sharing text editor. This is by no means the only use of such a search routine. For example, a variant of this algorithm is used as the symbol table search in an assembler.

It is assumed that the reader is familiar with regular expressions [2] and the machine language of the IBM 7094 computer [3].

The Compiler

The compiler consists of three concurrently running stages. The first stage is a syntax sieve that allows only syntactically correct regular expressions to pass. This stage also inserts the operator "·" for juxtaposition of regular expressions. The second stage converts the regular expression to reverse Polish form. The third stage is the object code producer. The first two stages are straightforward and are not discussed. The third stage expects a syntactically correct, reverse Polish regular expression.

The regular expression $a(b \mid c)*d$ will be carried through as an example. This expression is translated into $abc \mid * \cdot d \cdot$ by the first two stages. A functional description of the third stage of the compiler follows:

The heart of the third stage is a pushdown stack. Each entry in the pushdown stack is a pointer to the compiled code of an operand. When a binary operator ("|" or "·") is compiled, the top (most recent) two entries on the stack are combined and a resultant pointer for the operation replaces the two stack entries. The result of the binary operator is then available as an operand in another operation. Similarly, a unary operator ("*") operates on the top entry of the stack and creates an operand to replace that entry. When the entire regular expression is compiled, there is just one entry in the stack, and that is a pointer to the code for the regular expression.

The compiled code invokes one of two functional routines. The first is called NNODE. NNODE matches a single character and will be represented by an oval containing the character that is recognized. The second functional routine is called CNODE. CNODE will split the

current search path. It is represented by ⊕ with one input path and two output paths.

Figure 1 shows the functions of the third stage of the compiler in translating the example regular expression. The first three characters of the example *a*, *b*, *c*, each create a stack entry, $S[i]$, and an NNODE box.

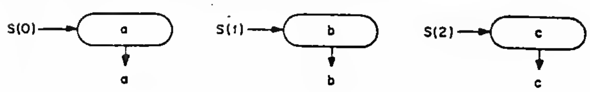

Fig. 1

The next character "*" combines the operands *b* and *c* with a CNODE to form *b*|*c* as an operand. (See Figure 2.)

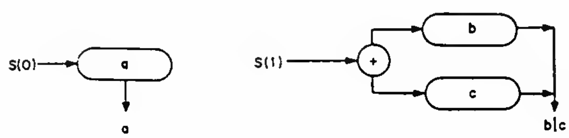

Fig. 2

The next character "*" operates on the top entry on the stack. The closure operator is realized with a CNODE by noting the identity $X* = \lambda XX*$, where X is any regular expression (operand) and λ is the null regular expression. (See Figure 3.)

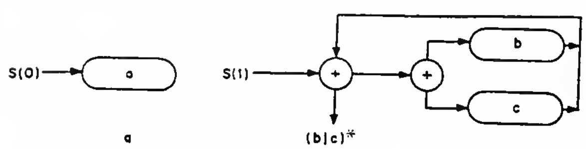

Fig. 3

The next character "·" compiles no code, but just combines the top two entries on the stack to be executed sequentially. The stack now points to the single operand $a \cdot (b|c)*$. (See Figure 4.)

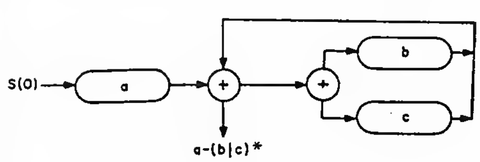

Fig. 4

The final two characters *d*· compile and connect an NNODE onto the existing code to produce the final regular expression in the only stack entry. (See Figure 5.)

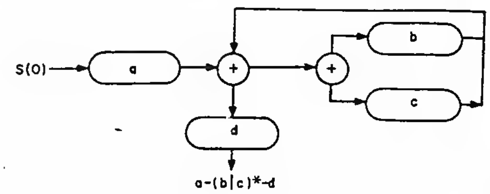

Fig. 5

A working example of the third stage of the compiler appears below. It is written in ALGOL-60 and produces object programs in IBM 7094 machine language.

```
begin
  integer procedure get character;  code;
  integer procedure instruction(op, address, tag, decrement);
    code;
  integer procedure value(symbol);  code;
  integer procedure index(character);  code;
  integer char, lc, pc;
  integer array stack[0:10], code[0:300];
  switch switch := alpha, juxta, closure, or, eof;
  lc := pc := 0;
advance:
  char := get character;
  go to switch[index(char)];
alpha:
  code[pc] := instruction('tra', value('code')+pc+1, 0, 0);
  code[pc+1] := instruction('txl', value('fail'), 1, -char-1);
  code[pc+2] := instruction('txh', value('fail'), 1, -char);
  code[pc+3] := instruction('tsx', value('nnode'), 4, 0);
  stack[lc] := pc;
  pc := pc+4;
  lc := lc+1;
  go to advance;
juxta:
  lc := lc-1;
  go to advance;
closure:
  code[pc] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+1] := code[stack[lc-1]];
  code[stack[lc-1]] := instruction('tra', value('code')+pc, 0, 0);
  pc := pc+2;
  go to advance;
or:
  code[pc] := instruction('tra', value('code')+pc+4, 0, 0);
  code[pc+1] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+2] := code[stack[lc-1]];
  code[pc+3] := code[stack[lc-2]];
  code[stack[lc-2]] := instruction('tra', value('code')+pc+1, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+4, 0, 0);
  pc := pc+4;
  lc := lc-1;
  go to advance;
eof:
  code[pc] := instruction('tra', value('found'), 0, 0);
  pc := pc+1
end
```

The integer procedure *get character* returns the next character from the second stage of the compiler. The

integer procedure *index* returns an integer index to classify the character. The integer procedure *value* returns the location of a named subroutine. It is an assembler symbol table routine. The integer procedure *instruction* returns an assembled 7094 instruction.

When the compiler receives the example regular expression, the following 7094 code is produced:

```
CODE    TRA   CODE+1          0    a
        TXL   FAIL,1,-'a'-1   1
        TXH   FAIL,1,-'a'     2
        TSX   NNODE,4         3
        TRA   CODE+16         4    b
        TXL   FAIL,1,-'b'-1   5
        TXH   FAIL,1,-'b'     6
        TSX   NNODE,4         7
        TRA   CODE+16         8    c
        TXL   FAIL,1,-'c'-1   9
        TXH   FAIL,1,-'c'     10
        TSX   NNODE,4         11
        TRA   CODE+16         12   |
        TSX   CNODE,4         13
        TRA   CODE+9          14
        TRA   CODE+5          15
        TSX   CNODE,4         16   *
        TRA   CODE+13         17
        TRA   CODE+19         18   ·d
        TXL   FAIL,1,-'d'-1   19
        TXH   FAIL,1,-'d'     20
        TSX   NNODE,4         21
        TRA   FOUND           22   ·eof
```

Runtime Routines

During execution of the code produced by the compiler, two lists (named CLIST and NLIST) are maintained by the subroutines CNODE and NNODE. CLIST contains a list of TSX **,2 instructions terminated by a TRA XCHG. Each TSX represents a partial match of the regular expression and the TRA XCHG represents the end of the list of possible matches. A call to CNODE from location x moves the TRA XCHG instruction down one location in CLIST and inserts in its place a TSX $x+1$,2 instruction. Control is then returned to $x+2$. This effectively branches the current search path. The path at $x+1$ is deferred until later while the branch at $x+2$ is searched immediately. The code for CNODE is as follows:

```
CNODE   AXC   **,7        CLIST COUNT
        CAL   CLIST,7
        SLW   CLIST+1,7   MOVE TRA XCHG
                            INSTRUCTION
        PCA   ,4
        ACL   TSXCMD
        SLW   CLIST,7     INSERT NEW TSX **,2
                            INSTRUCTION
        TXI   *+1,7,-1
        SCA   CNODE,7     INCREMENT CLIST
                            COUNT
        TRA   2,4         RETURN
*
TSXCMD  TSX   1,2         CONSTANT, NOT
                            EXECUTED
```

The subroutine NNODE is called after a successful match of the current character. This routine, when called from location x, places a TSX $x+1$,2 in NLIST. It then returns to the next instruction in CLIST. This sets up the place in CODE to be executed with the next character. The code for NNODE is as follows:

```
NNODE   AXC   **,7        NLIST COUNT
        PCA   ,4
        ACL   TSXCMD
        SLW   NLIST,7     PLACE NEW TSX **,2
                            INSTRUCTION
        TXI   *+1,7,-1
        SCA   NNODE,7     INCREMENT NLIST
                            COUNT
        TRA   1,2
```

The routine FAIL simply returns to the next entry in the current list CLIST.

```
FAIL    TRA   1,2
```

The routine XCHG is transferred to when the current list is exhausted. This routine copies NLIST onto CLIST, appends a TRA XCHG instruction, gets a new character in index register one, and transfers to CLIST. The instruction TSX CODE,2 is also executed to start a new search of the entire regular expression with each character. Thus the regular expression will be found anywhere in the text to be searched. Variations can be easily incorporated. The code for XCHG is:

```
XCHG    LAC   NNODE,7     PICK UP NLIST COUNT
        AXC   0,6         PICK UP CLIST COUNT
X1      TXL   X2,7,0
        TXI   *+1,7,1
        CAL   NLIST,7
        SLW   CLIST,6     COPY NLIST ONTO CLIST
        TXI   X1,6,-1
X2      CLA   TRACMD
        SLW   CLIST,6     PUT TRA XCHG AT
                            BOTTOM
        SCA   CNODE,6     INITIALIZE CNODE
                            COUNT
        SCA   NNODE,0     INITIALIZE NNODE
                            COUNT
        TSX   GETCHA,4
        PAC   ,1          GET NEXT CHARACTER
        TSX   CODE,2      START SEARCH
        TRA   CLIST       FINISH SEARCH
*
TRACMD  TRA   XCHG        CONSTANT, NOT
                            EXECUTED
```

Initialization is required to set up the initial lists and start the first character.

```
INIT    SCA   NNODE,0
        TRA   XCHG
```

The routine FOUND is transferred to for each successful match of the entire regular expression. There is a one character delay between the end of a successful match and the transfer to FOUND. The null regular expression is found on the first character while one character regular expressions are found on the second character. This means that an extra (end of file) character must be put through

the code in order to obtain complete results. FOUND depends upon the use of the search routine and is therefore not discussed in detail.

The integer procedure GETCHA (called from XCHG) obtains the next character from the text to be searched. This character is right adjusted in the accumulator. GETCHA must also recognize the end of the text and terminate the search.

Notes

Code compiled for $a**$ will go into a loop due to the closure operator on an operand containing the null regular expression, λ. There are two ways out of this problem. The first is to not allow such an expression to get through the syntax sieve. In most practical applications, this would not be a serious restriction. The second way out is to recognize lambda separately in operands and remember the CODE location of the recognition of lambda. This means that $a*$ is compiled as a search for $\lambda|aa*$. If the closure operation is performed on an operand containing lambda, the instruction TRA FAIL is overlaid on that portion of the operand that recognizes lambda. Thus $a**$ is compiled as $\lambda|aa*(aa*)*$.

The array *lambda* is added to the third stage of the previous compiler. It contains zero if the corresponding operand does not contain λ. It contains the *code* location of the recognition of λ if the operand does contain λ. (The *code* location of the recognition of λ can never be zero.)

```
begin
  integer procedure get character;  code;
  integer procedure instruction(op, address, tag, decrement);
    code;
  integer procedure value(symbol);  code;
  integer procedure index(character);  code;
  integer char, lc, pc;
  integer array stack, lambda[0:10], code[0:300];
  switch switch := alpha, juxta, closure, or, eof;
  lc := pc := 0;
advance:
  char := get character;
  go to switch[index(char)];
alpha:
  code[pc] := instruction('tra', value('code')+pc+1, 0, 0);
  code[pc+1] := instruction('txl', value('fail'), 1, -char-1);
  code[pc+2] := instruction('txh', value('fail'), 1, -char);
  code[pc+3] := instruction('tsx', value('nnode'), 4, 0);
  stack[lc] := pc;
  lambda[lc] := 0;
  pc := pc+4;
  lc := lc+1;
  go to advance;
juxta:
  if lambda[lc-1] = 0 then
    lambda[lc-2] := 0;
  lc := lc-1;
  go to advance;
closure:
  code[pc] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+1] := code[stack[lc-1]];
  code[pc+2] := instruction('tra', value('code')+pc+6, 0, 0);
  code[pc+3] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+4] := code[stack[lc-1]];
  code[pc+5] := instruction('tra', value('code')+pc+6, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+3, 0, 0);
  if lambda[lc-1] ≠ 0 then
    code[lambda[lc-1]] := instruction('tra', value('fail'), 0, 0);
  lambda[lc-1] := pc+5;
  pc := pc+6;
  go to advance;
or:
  code[pc] := instruction('tra', value('code')+pc+4, 0, 0);
  code[pc+1] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+2] := code[stack[lc-1]];
  code[pc+3] := code[stack[lc-2]];
  code[stack[lc-2]] := instruction('tra', value('code')+pc+1, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+4, 0, 0)
  if lambda[lc-2] = 0 then
    begin if lambda[lc-1] ≠ 0 then
      lambda[lc-2] = lambda[lc-1]
    end else
    if lambda[lc-1] ≠ 0 then
      code[lambda[lc-1]] :=
        instruction('tra', value('code')+lambda[lc-2], 0, 0);
  pc := pc+4;
  lc := lc-1;
  go to advance;
eof:
  code[pc] := instruction('tra', value('found'), 0, 0);
  pc := pc+1
end
```

The next note on the implementation is that the sizes of the two runtime lists can grow quite large. For example, the expression $a*a*a*a*a*$ explodes when it encounters a few concurrent a's. This expression is equivalent to $a*$ and therefore should not generate so many entries. Such redundant searches can be easily terminated by having NNODE (CNODE) search NLIST (CLIST) for a matching entry before it puts an entry in the list. This now gives a maximum size on the number of entries that can be in the lists. The maximum number of entries that can be in CLIST is the number of TSX CNODE,4 and TSX NNODE,4 instructions compiled. The maximum number of entries in NLIST is just the number of TSX NNODE,4 instructions compiled. In practice, these maxima are never met.

The execution is so fast, that any other recognition and deleting of redundant searches, such as described by Kuno and Oettinger [4], would probably waste time.

This compiling scheme is very amenable to the extension of the regular expressions recognized. Special characters can be introduced to match special situations or sequences. Examples include: beginning of line character, end of line character, any character, alphabetic character, any number of spaces character, lambda, etc. It is also easy to incorporate new operators in the regular expression routine. Examples include: not, exclusive or, intersection, etc.

REFERENCES

1. Brzozowski, J. A. Derivatives of regular expressions. *J. ACM 11*, 4 (Oct. 1964), 481–494.
2. Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata Studies*, Ann. Math. Stud. No. 34. Princeton U. Press, Princeton, N.J., 1956, pp. 3–41.
3. IBM Corp. IBM 7094 principles of operation. File No. 7094-01, Form A22-6703-1.
4. Kuno, S., and Oettinger, A. G. Multiple-path syntactic analyzer. Proc. IFIP Congress, Munich, 1962, North-Holland Pub. Co., Amsterdam.